

4.1 Suffix Tree

Suffix tree is a fundamental data structure for combinatorial pattern matching[1]. In this section, we will give a brief introduction to suffix tree. First, we define the suffix tree. Then we describe Ukkonen's algorithm to construct the suffix tree. At last we show some simple applications of it.

4.1.1 Suffix Trie

Suffix trie is a tree containing all possible suffices of a string S where each edge is labelled by a character. For example, let $S = acacag\$$, where $\$$ means the end of S . All possible suffices are shown in Table 4.1.

	Suffix
1	$acacag\$$
2	$cacag\$$
3	$acag\$$
4	$cag\$$
5	$ag\$$
6	$g\$$
7	$\$$

Table 4.1: Suffices of String S

The suffix trie built according to the former table is shown in Figure 4.1.

Every possible suffix is a path from the root to the leaf.

4.1.2 Suffix Tree

1. Definition:

A suffix tree is the compacted trie of all the suffices of a string S , merging nodes with only one child. The suffix tree made from the former suffix trie is shown in Figure 4.2. In this figure, the path-label of the node V is "aca";

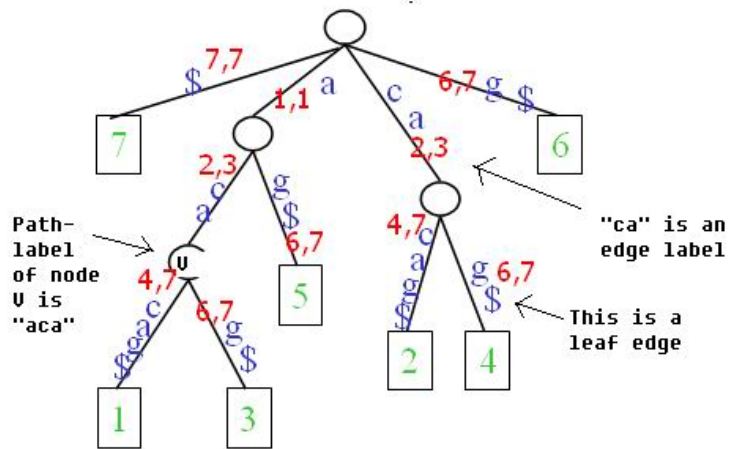


Figure 4.2: A suffix tree

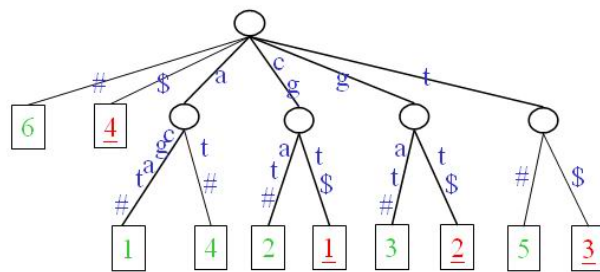


Figure 4.3: The generalized suffix tree

An occurrence of a substring Q in a string S is a list of all the positions where Q starts. For example, in string $S = acacag\$$, we want to find the occurrence of substring $Q = aca\$$. It is obvious that the start place of suffix $acacag\$$ is the start place of Q ; the start place of $acag\$$ is the start place of Q too. So if we find out the occurrence of $acacag\$$ and $acag\$$, we can find out the occurrence of Q . The occurrences can be found easily using a suffix tree. For time execution, Since a node can be found by traversing at most $|Q|$ edges and $\text{occ}(\text{total number of occurrences})$ leaves are visited, so the time complexity is $O(|Q| + \text{occ})$.

2. Find the longest repeated substring in S .

In order to find the longest repeated substring in S , we just need to find the deepest internal node in the suffix tree of S . For example, the longest repeated substring of $S = acacag$ can be found by locating the deepest internal node in the suffix tree of S as show in Figure 4.6. From the figure, the deepest internal node in the tree is the father of leaf 3 and leaf 1, so the longest repeated substring is from the root to this node: aca .

3. Find the longest common substring of two or more sequences.

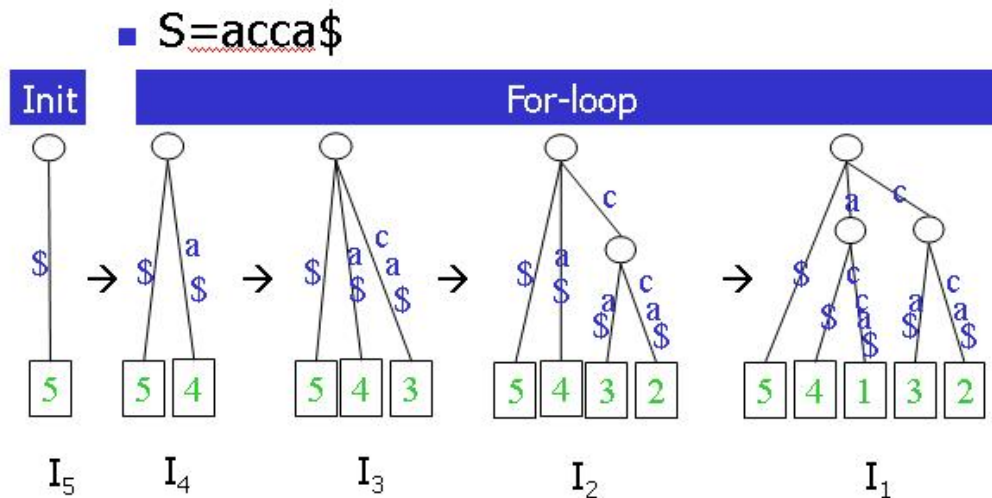
For example, consider two strings S_1 and S_2 , we can find the longest common substring of S_1 and S_2 by utilizing the generalized suffix tree for S_1 and S_2 as follows: First, build generalized suffix tree for $S_1\#$ and $S_2\$$; Then, mark each internal node with leaves representing suffices of both S_1 and S_2 . Finally, report the deepest marked node.

4.3 Construction of Suffix Tree

In this section, the algorithm for constructing the suffix tree is presented. In fact, a suffix tree can be constructed in $O(n)$ time. Weiner's algorithm (1973)[1] requires linear time for constant size alphabet but much space is needed. McCreight's algorithm (1976)[2] requires linear time for constant size alphabet and quadratic space. Later in 1995, Ukkonen presented an online algorithm which requires linear time for constant size alphabet.

Farach's algorithm (1997) [4] shows that even for general alphabet, suffix tree can be constructed in linear time.

This section presents a detailed algorithm to construct the suffix tree for $S = s_1s_2..s_n$ where $s_n = \$$.

Figure 4.4: Example of constructing suffix tree for $S = \text{acca}\$$

First, we present a straightforward algorithm, whose time complexity is $O(n^2)$. We initialize the tree with only a root and include suffices into the tree one by one for $i = n$ to 1. Though such algorithm is straightforward, the time is $O(n^2)$, which is costly. Figures 4.4 and 4.5 show an example for constructing suffix tree for $S = \text{acca}\$$, and an example for constructing generalized suffix tree for $S = \text{acca}\$$ and $S' = c\#$.

Now we introduce Ukkonen's Algorithm[3]. First, we should define the concept of Implicit suffix tree.

An implicit suffix tree is obtained by removing '\$' from edge labels first. Then, edges with no label are removed, at last we contract nodes with only one child. An example is given in Figure 4.6.

Ukkonen's Algorithm[3] is described as follow.

Denote I_i be the implicit suffix tree for $S[1..i]$. First we construct I_1 . It is easy and explicit to do so. Then the idea is to construct I_{i+1} from I_i for $i = 1$ to $m - 1$. The process is illustrated in Figure 4.7. For phase i (constructing I_{i+1} from I_i), there are $i + 1$ extensions. Each extension is done by locating the end of the path from the root labelled $S[j..i]$ and extends the path with character $S[i + 1]$. Such extension should obey one of the following 3 rules:

1. Rule 1: If $S[j..i]$ ends at a leaf, $S[i + 1]$ is appended to the end of the label

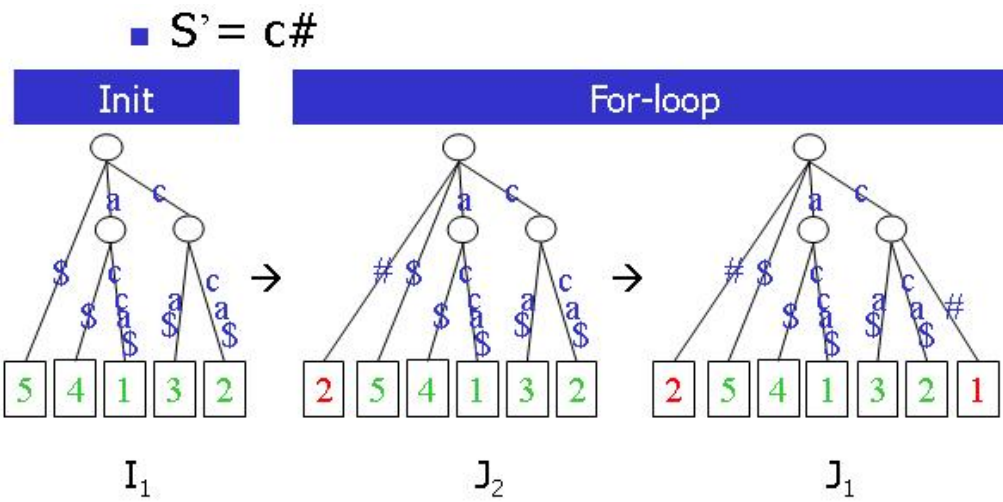


Figure 4.5: Example of constructing generalized suffix tree for $S = acca\$$ and $S' = c\#$

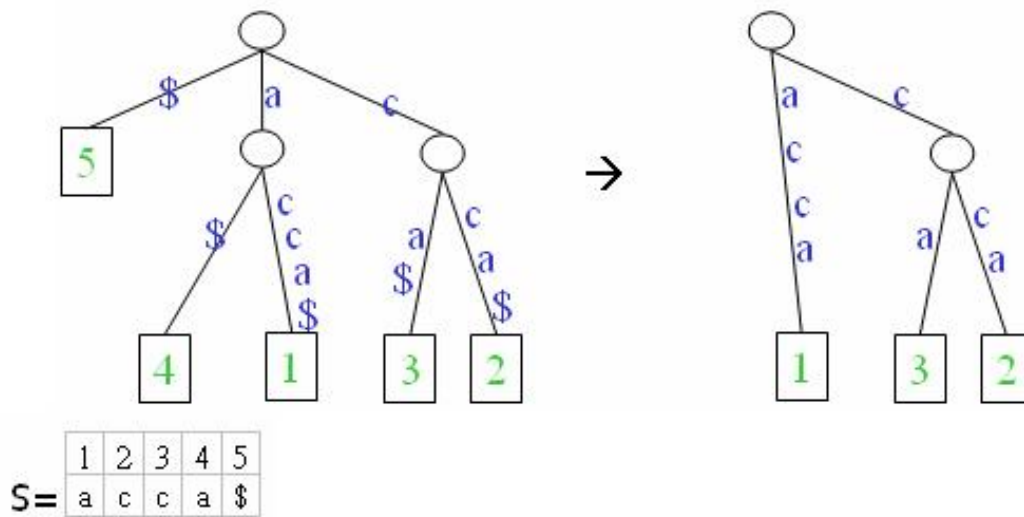


Figure 4.6: Obtain Implicit Suffix Trie

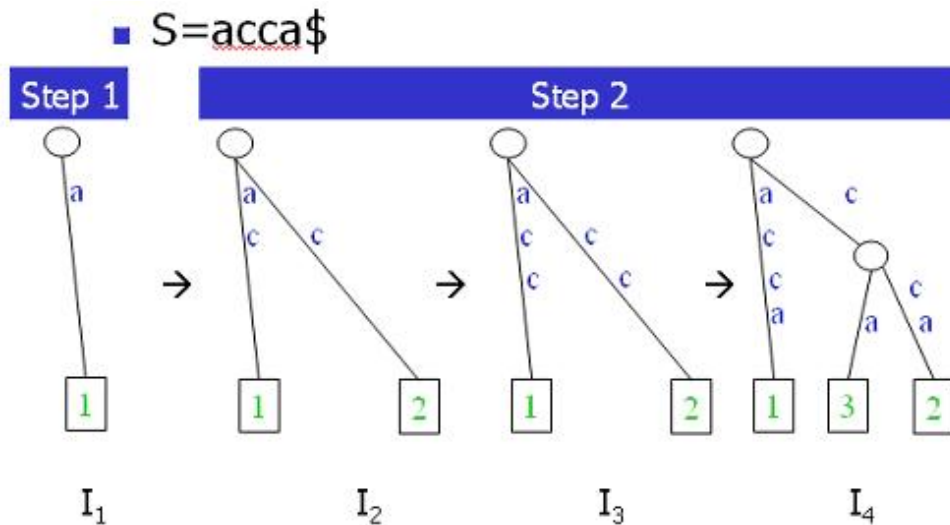


Figure 4.7: Illustration of the Whole Algorithm

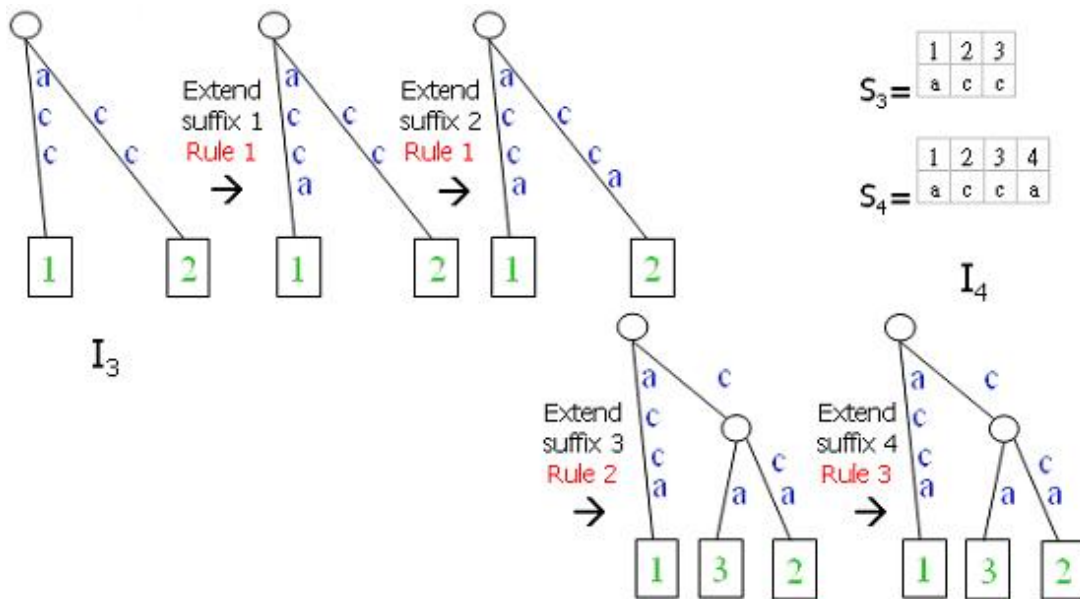
on the leaf edge.

2. Rule 2: If no path from the end of $S[j..i]$ starts with character $S[i+1]$, create a new leaf edge labelled with character $S[i+1]$. This is the only extension that will increase the number of leaves in the tree.
3. Rule 3: If some path from the end of $S[j..i]$ starts with character $S[i+1]$, then we do nothing on the tree. The reason is that the ending $S[i+1]$ already appears.

The process of constructing I_4 from I_3 (phase 3) for $S = \text{acca}\$$ is illustrated in Figure 4.8.

When we evaluate Phase i , the process of constructing I_{i+1} from I_i , we can conclude several observations. Suppose rule 3 is applied to extension j , the path labelled $S[j..i]$ in the current tree I_i must continue with character $S[i+1]$. That means the subtree $S[j..i+1]$ is already in the tree. So for all $j' \geq j$, the subtree $S[j'..i+1]$ is already in the tree. It implies that, for $j' \geq j$, the path labelled $S[j'..i]$ also continue with character $S[i+1]$. Thus rule 3 will be applied for extension j' . Therefore, if we apply rule 3 for extension j , then rule 3 will be applied for extension j' for $j' = j, \dots, i+1$. Thus, we need to do nothing for $j' = j+1, \dots, i+1$. Conclusively, once we apply rule 3 for some extension j in Phase i , we can stop. Such conclusion reduce the cost of the work.

Another observation is that once we form a leaf for a suffix in I_i , the leaf appears in $I_{i'}$ for $i' > i$. The reason is that there is no rule for removing leaves when we trace the process of the construction. So such observation is correct.

Figure 4.8: Illustration the construction of I_4 from I_3

Based on previous observations, we can reach a rather useful conclusion: In Phase i , let j_i be the last extension using rule 1 or rule 2. In other words, extension j' is based on rule 3 for $j' > j_i$. Then, in Phase $i + 1$, extension j' for $j' \leq j_i$ is based on rule 1. Furthermore, Extension j' for $j' > j_{i-1}$, cannot be based on rule 1.

Then we can improve the algorithm. For phase i , when doing extension j for $j \in [1, j_{i-1}]$, these extensions are based on rule 1. So we can do nothing. When $j \in [j_{i-1}, i + 1]$, we first find the end of the path labelled $\beta = S[j..i]$ and extend it with character $S[i + 1]$ by applying rule 2 or 3. If the extension is by rule 3, it means that it is the end of this phase. Thus, we only need to set $j_i = j - 1$ and end phase i .

Construction using this algorithm improves the whole process. Detailed description is showed below:

- Phase 1: we compute extension $1..j_1 + 1$.
- Phase 2: we compute extension $j_1 + 1..j_2 + 1$.
- Phase 3: we compute extension $j_2 + 1..j_3 + 1$.
- ...
- Phase i : we compute extension $j_{i-1} + 1..j_i + 1$.
- ...

So, we do at most $2n$ extensions in all phases. Because for each extension j , it takes $O(n)$ time to find the path labelled $S[j..i]$. Thus, the total time is $O(n^2)$.

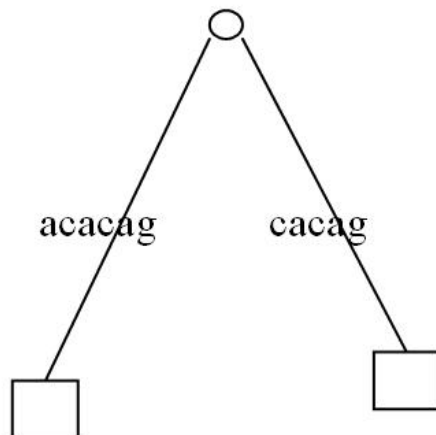


Figure 4.9: Suffix Tree After Extension 2 in Phase 4

In the next section, the process can be further accelerated using suffix link.

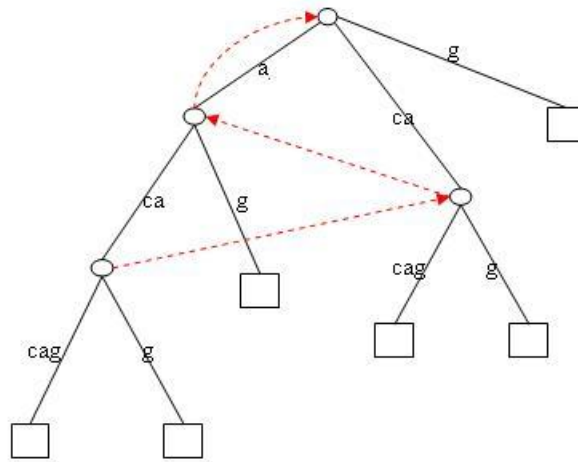
4.4 Suffix Link

4.4.1 Idea of Suffix Link

Though we can greatly reduce work by the above two observations, the extension steps based on rule 2 and rule 3 in each phase are still time-consuming. In Phase i for Extension $j > j_{i-1}$, we need to locate $s[j..i]$ by walking down the tree from the root, and then determine whether the extension should be based on rule 2 or rule 3. Walking down the tree is very time consuming. For example, suppose we are doing Extension 3 in Phase 5 for the string “acacag”. Figure 4.9 shows the suffix tree after Phase 4. At that time, $j_4 = 2$.

In Phase $i = 5$, Extension $(j_{i-1} + 1) = 3$, we have to locate $s[j_{i-1} + 1..5] = “aca”$ by walking down the tree from the root, and check whether the suffix “acag” should be added to the tree or already exists in the tree. Since “acag” does not exist in the tree, we add the extension based on rule 2. Then, we have to check the suffix “cag” to see if it is still based on rule 2. Similarly, we walk down the tree from the root, check the suffices “cag”, “ag” and “g”, and then apply rule 2 to them.

To speed up the process, we make the third important observation on the suffices needed to be checked in a phase which rule is suitable to it. In the above example, these suffices are “acag”, “cag”, “ag” and “g”, in which the neighbor

Figure 4.10: Suffix Tree and Suffix Links of *acacag*

suffixes share the same suffix except that the latter suffix misses the first letter when compared to the former suffix. In other words, there is some relationship between these suffixes. Below we introduce a new data structure suffix link, which takes advantage of the relationship to speed up the extensions.

4.4.2 Definition of Suffix Link

Suffix link is defined as follows: For an internal node v with path-label $x\alpha$, if there is another node $s(v)$ with path-label α , then we create a suffix link from v to $s(v)$.

Lemma 4.1 *For an implicit suffix tree T_i for $s[1..i]$, each internal node must have a suffix link, assuming that the root has a suffix link to itself.*

Proof: Consider any internal node v of T_i with the path-label $x\alpha$, which is the common prefix of $s[j..i]$ and $s[j'..i]$. So α is the common prefix of $s[j+1..i]$ and $s[j'+1..i]$. Thus, there exists an internal node v' of T_i with the path-label α . We have a suffix link from v to v' . The lemma follows. ■

For every internal node v , if we represent a suffix link using a dashed line, from v to $s(v)$, Figure 4.10 shows the suffix links for internal nodes in the implicit suffix tree of “*acacag*”.

4.4.3 Creation, Maintenance and Taking Advantage of Suffix Link

Suffix links must be created first before they can speed up the extensions. Suppose an extension in a phase is based on rule 2, a new internal node will be created. However, the suffix link of this newly created node doesn't exist at that time. Therefore, we need a method to create the suffix links.

Assume we have maintained suffix links for all internal nodes before Phase i . At the beginning of Phase i , we are at position w , which is the end of $s[j_{i-1} + 1..i]$, where rule 2 or rule 3 is used last time. If rule 2 is used last time, w is an internal node. If rule 3 is used last time, w may be at the middle of an edge. Then, for each subsequent Extension j , we do the following steps:

Step (1): Check whether rule 2 or rule 3 should be applied in Extension j , and apply the corresponding rules to Extension j .

Step (2): Find the first internal node v at or above w , which has suffix link. (We assume that the root has a suffix link to itself)

Step (3): Go through the suffix link of v to $s(v)$

Step (4): Go down zero or more nodes until we find the end of $s[j + 1..i]$, says, w' .

Step (5): If $w \neq v$, it means w does not have a suffix link. Create a suffix link from w to w'

Step (6): If Extension j is the last extension based on rule 2 or is based on rule 3, set $j_i = j - 1$ and go to Phase $i + 1$. Otherwise, set $w = w'$, and go to the next extension, Extension $j + 1$.

Figure 4.11 shows how to perform the step (1) - (5).

Even if we have to spend time creating suffix links for newly created internal nodes, the existing suffix links still help to speed up the extensions. Without them, after we finish an extension, we have to return to the root and then locate a new position from the root. The existing suffix links prevent us from unnecessary going up and down.

Let's look at an example: Phase 5 of creating suffix links for the implicit suffix tree of "acacag". At the beginning of Phase $i = 5$, $j_{i-1} = 2$, we have already got the tree shown in Figure 4.12.

In Extension 3, we find the suffix "acag" does not exist in the tree, so we apply rule 2: split the edge "acacag" at point w , and create an internal node w and a new leaf, the edge between them is "g". Then we try to find the first internal node at or above w who has a suffix link, in this case, the root. We assume the

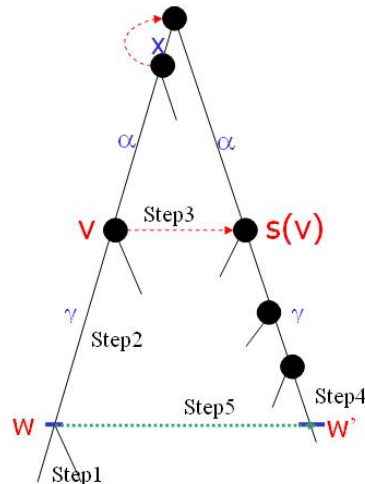


Figure 4.11: Step (1) - (5) of Creating Suffix Links

root has a suffix link to itself. Going through the suffix link of the root, we arrive at the root again. Then we go down several nodes and find point w' , which is the end of $s[j + 1..i] = s[4..5]$. Since w does not have a suffix link, we create a suffix link from w to w' . (See figure 4.12) Extension 3 is not the last extension of Phase 5, we set $w = w'$, and go on with Extension 4.

Extension 4 is quite similar to Extension 3. We apply rule 2, split the edge at w , create an internal node and a new leaf, and mark the new edge “ g ”. Then we go up and find the root as the first internal node who has a suffix link. Going through the suffix link and going down, we find the w' as the end of $s[5..5]$. Then we create a suffix link from w to w' , set $w = w'$, and go on with the next Extension. Figure 4.12 shows the result after Extension 4.

Extension 5 is also similar. Figure 4.12 shows the result after Extension 5.

Extension 6 is based on rule 2. Since w is the root in this extension, who has already a suffix link, no suffix link is created in this extension. See figure 4.12

Situation is better if the internal node w is located at an existing internal node who has already got a suffix link. Let us see another example: Suppose we are going to do the last phase of building the suffix tree of the string “*acacagacat*”, when $j_{i-1} = 6$ at that time. Figure 4.13 shows the suffix tree before Extension 6.

In this case, because w has already the suffix link, the first internal node at or above w is w itself. We go through the suffix link. We need not go down a

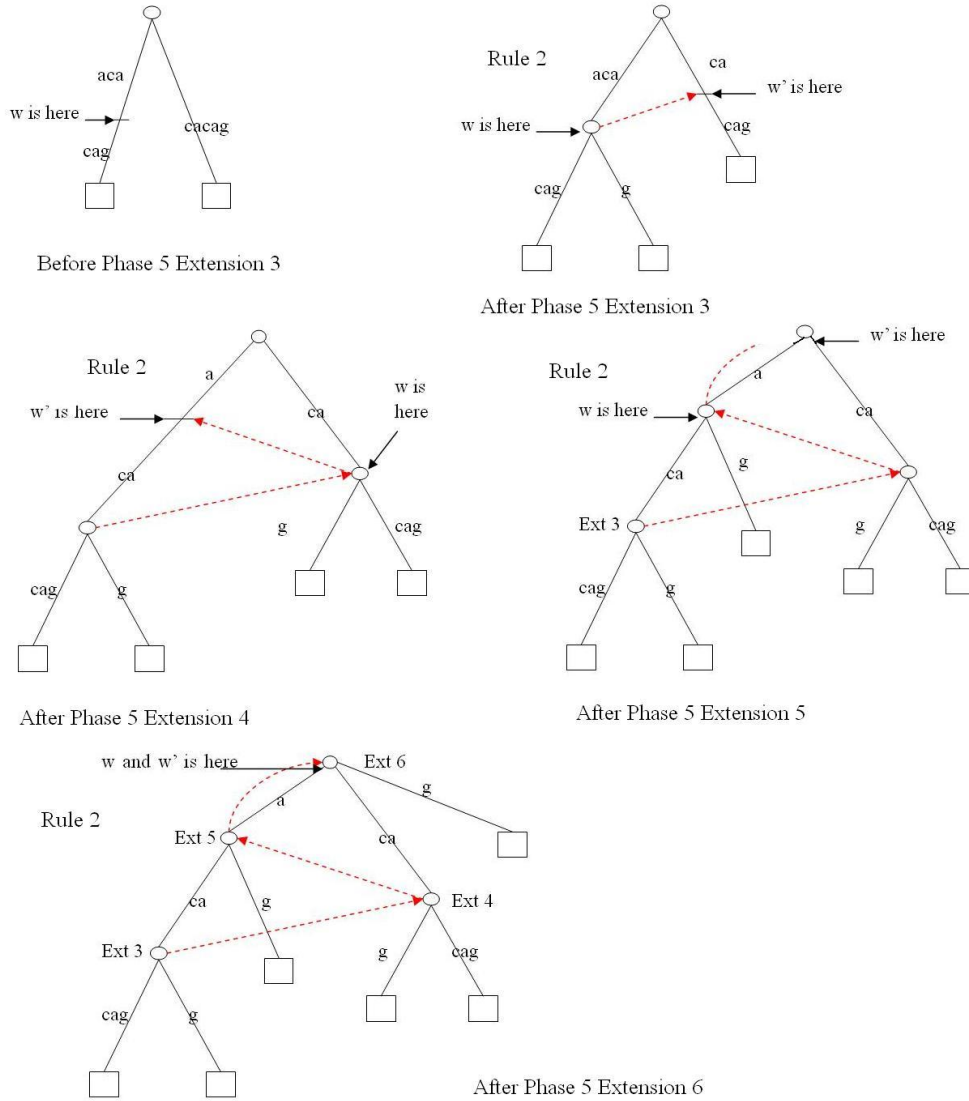
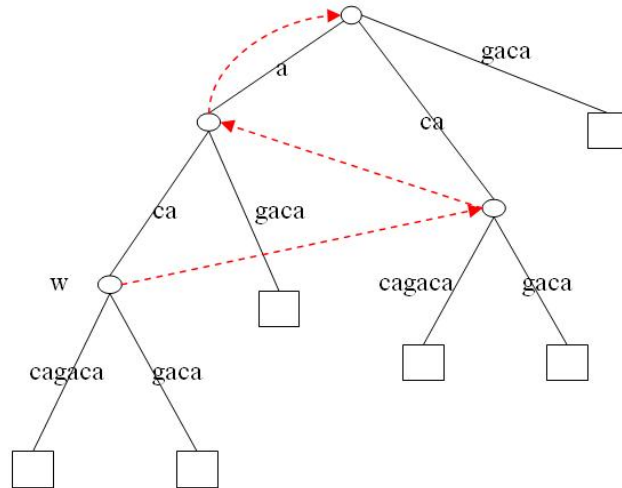


Figure 4.12: After Phase 5 Extension 6 for Suffix Tree of *acacag*

Figure 4.13: Before Extension 6 for Suffix Tree of *acacagacat*

number of nodes according to step (4) of creating suffix links. You can see from figure 4.14 that step (4) is really skipped.

4.4.4 Time Complexity

With the help of suffix links, the extension based on rule 2 or rule 3 are simplified. In the suffix link algorithm, step (1), (2), (3), (5) and (6) take $O(1)$ time; as for step (4), it takes amortized $O(1)$ time. So, each extension can be solved in $O(1)$ time. As there are $2n$ extensions, the total time is $O(n)$.

Why does step (4) takes amortized $O(1)$ time? Denote node depth to be the number of edges from the root. Then, in each extension, step (2) reduces the node-depth by at most 1. For every internal node v , node-depth of v is equal or less than $1 + \text{node-depth of } s(v)$. (see figure 4.15) Therefore, step (3) reduces the node-depth by at most 1. Since there are $2n$ extensions, all steps (2) and (3) can reduce the node-depth by at most $4n$. Since the maximum node-depth is n , all steps (4) can at most increase the node-depth by $5n$. By amortization, each step (4) go down amortized $O(1)$ nodes, which takes amortized $O(1)$ time.

In conclusion, with the help of the suffix link, we can build the implicit suffix tree in $O(n)$ time instead of $O(n^2)$ time.

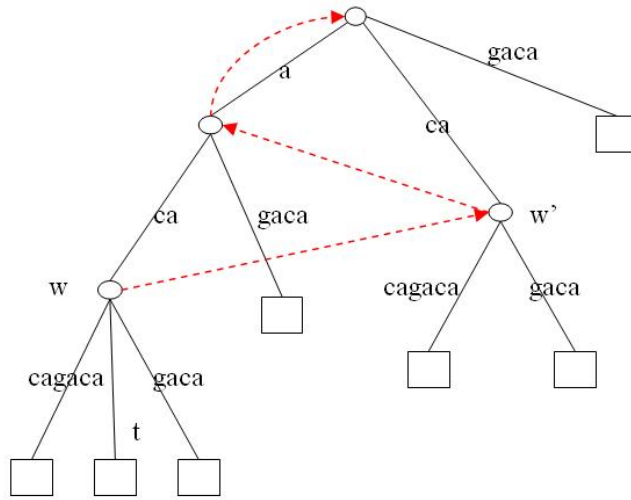


Figure 4.14: After Extension 6 for Suffix Tree of *acacagacat*

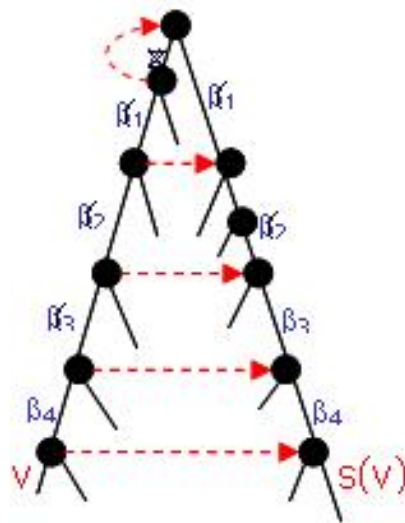


Figure 4.15: Time Complexity

4.4.5 From Implicit Suffix Tree to Explicit Suffix Tree

Suppose we have already got an implicit suffix tree for $s[1..n]$, we can convert it to an explicit suffix tree. First we add the character '\$' at the end of the string $s[1..n]$. In other words, let $s[n + 1] = '$'$. Then we perform Phase n, to add '\$' into the suffix tree.

Since '\$' is the last character of the string $s[1..n + 1]$, it is the common suffix for all the suffices in the string. Therefore it is also the last character on every edge connecting to a leaf in the explicit suffix tree. Since '\$' is a new character for the string, for any suffix in the implicit suffix tree, adding this character will not use rule 3. It will use rule 1 to append '\$' to each edge connecting to a leaf; and use rule 2 to split some edges and for each edge to be split create a new internal node and a new branch from the internal node labelled '\$'. This is done by $O(n)$ traversal of the tree. The result is an explicit suffix tree.

4.5 Applications of Suffix Tree

4.5.1 Exact String Matching Problem

Formalized task:

Given a string S of length n , for any query Q , find all occurrences of Q in S .

Simple example:

Given $S = acacag$, find all occurrences of $Q = aca$ in S .

Solution:

Preprocessing : Build a suffix tree for string S .

Query:

1. Search for the node x in the suffix tree which represent Q .
2. All the leaves in the subtree rooted at x are the occurrences.

Preprocessing takes $O(n)$ time. Each query takes $O(|Q| + occ)$ time where occ is the number of occurrences of Q in S .

4.5.2 Longest Repeated Substring Problem

Formalized task:

Given a string S of length n , find the longest repeated substring in S .

Simple example:

Given $S = acacag$, the longest repeated substring of S should be aca .

Solution:

1. Build a suffix tree for string S . This step takes $O(n)$ time
2. Search the suffix tree to find the deepest internal node. This step takes $O(n)$ time.

Time complexity is $O(n)$.

4.5.3 Longest Common Substring Problem**Formalized task:**

Given strings P_1, P_2, \dots, P_k , find the longest common substring of these strings. ($k > 1$) The total length of these strings is n .

Simple example:

Given $P_1 = acgat$, $P_2 = cgt$, the longest common substring of P_1 and P_2 is cg .

Solution:

1. Assign distinct ending character to each string.
2. Build a general suffix tree for all strings. This step takes $O(n)$ time.
3. Mark each internal node with leaves representing suffixes of all P_i . This step takes $O(n)$ time.
4. Report the deepest marked node. This step takes $O(1)$ time.

Time complexity: $O(n)$.

4.5.4 Remain of US military personnel

Task: To identify the US military identity when killed. From each live US military, a small interval of his DNA is extracted (which are likely to be unique for different people) and forms a database. When someone is killed, we can extract

his DNA from his remain. However, due to the condition of the remain, we may not be able to get the complete information of the desired DNA interval, and may only get a substring S . We find all strings in the database containing S as a substring to determine who it may be.

Formalization:

The remains of US military personnel can be formalized as the substring problem for a database of patterns. Given a set of strings (database) of total length is n . For any query string S , find all strings in the database containing S as a substring.

Solution:

Preprocessing: Build a generalized suffix tree T for all strings in the database.

Query: Find all occurrences of S in T .

Preprocessing takes $O(n)$ time while each query can be solved in $O(|S| + occ)$ time where occ is the number of occurrences of S .

4.5.5 DNA Contamination Problem

Task:

Contamination is a problem in bio engineering. Let S be a sequence reconstructed from a genome, says, *C.elegans*, S is suspected to be contaminated by cloning vectors, PCR primes, or genome of the host organism (e.g. yeast), how can we determine it?

Formalization:

Given a string S_1 (the sequenced string) and a string S_2 (the combined sources of possible contamination), find all substrings of S_2 that occur in S_1 which are longer than some given length x .

Solution:

1. Build a generalized suffix tree for S_1 and S_2 .
2. Mark each internal node with leaves representing suffixes of both S_1 and S_2 .
3. Report all marked nodes with string depth of x or greater.

Figure 4.16 shows a simple example of this solution.

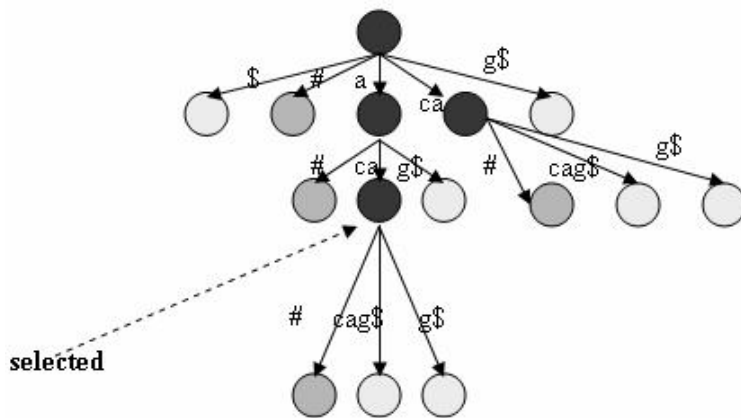


Figure 4.16: Example of DNA contamination problem. For $S_1 = acacag\$$, $S_2 = aca\#$, the set of substrings of S_2 of length > 2 that occur in $S_1 = \{aca\}$

Time complexity is $O(n)$.

4.5.6 Common Substring of more than 2 Strings

Task:

Given a set of strings (protein or DNA sequences), we want to know what substrings are common to a large number of these strings?

Formalization:

Consider K strings whose total length is n . For all $2 \leq k \leq K$, compute $l(k)$ which is the length of the longest substring common to at least k of these strings.

Simple example:

Consider a set of 5 strings { sandollar, sandlot, handler, grand, pantry } Then, when $K = 4$ and $l(k)$ is 3, the corresponding substring is *and*.

Solution:

1. Assign distinct ending character to each string.
2. Build a general suffix tree T for all strings. This step takes $O(n)$ time.
3. By traversing T , for each internal node v , compute its string depth. This step takes $O(n)$ time.

4. By traversing T , for each internal node v , compute $C(v)$. $C(v)$ is defined as the number of distinct termination symbols in the subtree rooted at v . This step takes $O(Kn)$ time.
5. Traverse T and visit every internal node v . For each v , if $V(C(v)) < \text{string-depth of } v$, set $V(C(v)) = \text{string-depth of } v$.
6. $l(k) = V(K)$, $l(k) = \max\{l(k), V(k+1)\}$ for $k = 2, \dots, k-1$. This step and step (5) takes $O(n)$ time.

Time complexity is $O(Kn)$.

4.5.7 Longest Common Prefix

Formalized task:

Given a string S of length n , for any i, j , find the length of the longest common prefix of suffixes i and j of S .

Solution:

The basic idea is that the lowest common ancestor of leaves i and j identifies the longest common prefix. The solution for longest common prefix is first obtained by Harel and Tarjan [5], and later simplified by Schieber and Vishkin [6].

1. Build suffix tree for string S . This step takes $O(n)$ time.
2. Return prefix of suffix i and j . This step only takes $O(1)$ time.

Time complexity is $O(n)$ for preprocessing and $O(1)$ for each processing of suffix i and suffix j .

4.5.8 Finding Palindrome

Palindrome: Given a string S , a palindrome is a substring u of S , such that $u = u^r$, where u^r denotes the reverse string of u . E.g. $ACAGACA$ is a palindrome as $ACAGACA = (ACAGACA)^r$.

1. If $S[i..i+k-1] = S^r[n-i+1..n-i+k]$, then $u = S[i-k+1..i+k-1]$ is an odd length palindrome; (refer to Figure 4.17)
2. If $S[i..i+k-1] = S^r[n-i+2..n-i+k+1]$, then $u = S[i-k..i+k-1]$ is an even length palindrome. (refer to Figure 4.18)

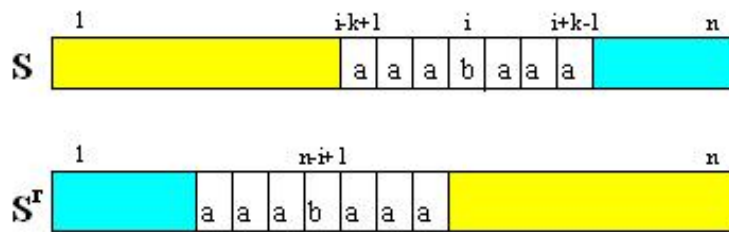


Figure 4.17: odd length palindrome

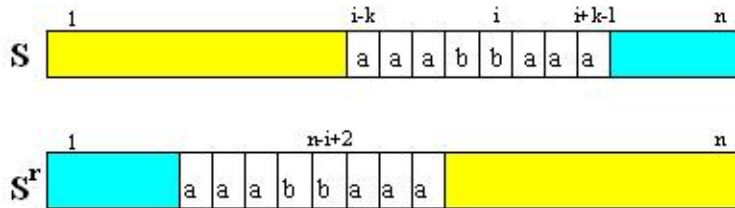


Figure 4.18: even length palindrome

Complemented palindrome: Given a string S , a complemented palindrome is a substring u of S , such that $u = \bar{u}^r$, where \bar{u} is complement of u , e.g. $ACAUGU$ is a complemented palindrome.

Maximal palindrome: Given a string S , a palindrome $u = S[i..i + |U| - 1]$ is called a maximal palindrome of S , if $S[i'..j']$ is not a palindrome for any $[i'..j'] \supset [i..i + |U| - 1]$.

With this definition of maximal palindrome, every palindrome is contained in a maximal palindrome. In other word, maximal palindromes are a compact way to represent all palindromes.

We can define maximal complemented palindrome similarly.

Formalized task:

1. Given a sting S of length n , which represents the genome, the problem is to locate all maximal palindromes in S .
2. Given a sting S of length n , which represents the genome, the problem is to locate all maximal complemented palindromes in S .

In this section, we only give solution for Task (1), which is to find maximal palindromes. Note that maximal complemented palindromes can be found similarly.

Solution: Preprocess S and S^r in $O(1)$ time so that any longest common prefix query can be answered in constant time;

For $i=1$ to n ,

1. Find the longest common prefix for (S_i, S_{n-i+1}^r) in $O(1)$ time. If the longest prefix is k , we find an odd length maximal palindrome $S[i-k+1..i+k-1]$.
2. Find the longest common prefix for (S_i, S_{n-i+2}^r) in $O(1)$ time. If the longest prefix is k , we find an even length maximal palindrome $S[i-k..i+k-1]$.

Analysis:

The preprocessing takes $O(n)$ time, and each longest common prefix query can be found in $O(1)$ time. So we can find the maximal palindromes in $O(n)$ time.

4.6 Suffix Array

The suffix tree, although is a powerful data structure, has very limited usage in practice. This is mainly due to its large space requirement of order $O(n|\Sigma| \log n)$ bits. To solve this problem, Manber and Myers [7] proposed a new data structure in 1993, called suffix array, which has a similar functionality as suffix tree but only requires $O(n \log n)$ bits space.

4.6.1 Definition of Suffix Array

Let $S[1..n] = S[0]S[2] \dots S[n]$ be a string of length n over an alphabet Σ . We assume that $S[n] = \$$ is a unique terminator which is alphabetically smaller than other characters. For any $i = 1, 2, \dots, n$, $S[i..n]$ is a suffix of S . The suffix array $SA[1..n]$ of S is an array of integers such that $S[SA[i]..n]$ is lexicographically the i -th smallest suffix of S .

In other words, a suffix array is an array of indices to lexicographically sorted suffixes of string S . For example, consider $S = acacag\$$. Its suffix array is shown in Figure 4.19 below.

Suffix	Position
<i>acacag</i> \$	1
<i>cacag</i> \$	2
<i>acag</i> \$	3
<i>cag</i> \$	4
<i>ag</i> \$	5
<i>g</i> \$	6
\$	7

i	$SA[i]$	Suffix
1	7	\$
2	1	<i>acacag</i> \$
3	3	<i>acag</i> \$
4	5	<i>ag</i> \$
5	2	<i>cacag</i> \$
6	4	<i>cag</i> \$
7	6	<i>g</i> \$

Figure 4.19: suffixes and suffix array

Note that each index can be stored in $O(\log n)$ bits. Thus, the whole suffix array takes $O(n \log n)$ bits.

4.6.2 Construction of Suffix Array

Observe that the leaves of a suffix tree, if counted in lexicographical depth-first order, form the suffix array of that string. This is shown in Figure 4.20.

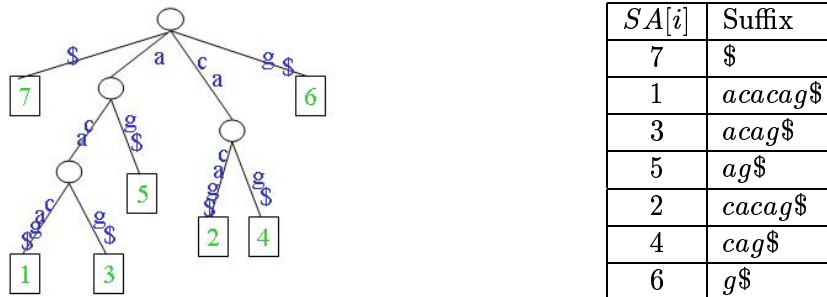


Figure 4.20: suffix tree and suffix array

The suffix tree T of $S[1..n]$ can be constructed in $O(n)$ time. Therefore, a naive way of suffix array construction is to first construct the suffix tree T , and then by lexicographically depth-first traversing T (of $O(n)$ time complexity), the suffix array SA of S is obtained. This takes $O(n)$ time in total.

However, this approach takes large working space, for the suffix tree requires $O(n|\Sigma| \log n)$ bits space. If we only have $O(n)$ bits working memory, the best known result for constructing suffix array takes $O(n)$ time [12].

4.6.3 Application

Find occurrence of query Q in a string S using suffix array.

Input:

1. The suffix array of a string S of length n
2. A query Q of length m .

Aim: Find all occurrences of Q in S .

Solution: The idea is to perform a binary search on the suffix array of S .

1. Let L and R be the left and right boundaries of SA such that $\text{suffix}_{SA[L]} \leq Q \leq \text{suffix}_{SA[R]}$ (initially, $L = 1$ and $R = n + 1$)
 Let l and r be the longest common prefix of Q and $\text{suffix}_{SA[L]}$ and $\text{suffix}_{SA[R]}$.
 Let $mlr = \min(l, r)$

2. Let $M = (L + R)/2$

We know that the first mlr characters of $\text{suffix}_{SA[M]}$ and Q are the same. Starting from the position mlr , find the last match position m . Find the lex-order of suffix_M and Q by comparing $\text{suffix}_{SA[M]}[m + 1]$ and $Q[m + 1]$. If $\text{suffix}_{SA[M]} > Q$, set $R = M$ and $r = m$ else set $L = M$ and $l = m$.

3. Goto 1.

The binary search performs at most $\log n$ comparisons and each comparison takes at most $O(m)$ time. Therefore, in the worst case, the algorithm takes $O(m \log n)$ time. It is reported that in practice, the actual time complexity is $O(m + \log n)$.

In the above section we show one example of replacing suffix tree by suffix array. In fact, most applications related to suffix tree can be solved using suffix array, with some additional time. When space is limited, suffix array is a good alternative to suffix tree.

4.7 FM-index

Although the space requirement of suffix array is less than that of suffix tree, it is still unacceptable in many real life situations. For example, the human genome, with its 3G base pairs, can take up to 40 Gigabytes storage for suffix tree and 13 Gigabytes for suffix array, which is clearly beyond the capacity of a normal PC. More space-efficient storage is required for practical situation.

Two such alternatives are proposed. The first is Compressed Suffix Array by Gross and Vitter [9]. Another is FM-index by Ferragine and Manzini [10]. Both of them can be stored in $O(n)$ bits. This means the human genome can be stored within 2 Gigabytes.

In the following section we are going to introduce the FM-index data structure.

4.7.1 Definition of FM-index

FM-index is a combination of Burrows-Wheeler (BW) text [11] and augmented structure to enable random access of the compressed data.

Let $T[1..n] = T[1]T[2] \dots T[n]$ be a string of length n , and $SA[n]$ be its suffix array, FM-index stores the following three data structures:

1. The BW is a string of characters defined as

$$BW[i] = \begin{cases} T[SA[i] - 1] & (\text{if } SA[i] \neq 1) \\ T[n] & (\text{if } SA[i] = 1) \end{cases}$$

2. $C[x]$ array stores the total number of occurrences of each character less than x .

3. A data structure which supports $O(1)$ computation of $occ(x, 1, i)$, with $occ(x, 1, i) = \text{number of occurrences of } x \text{ in } BW[1..i]$. The details of this data structure is omitted.

For example, consider $T = acacag\$$. Its BW text is shown in Figure 4.21, and $C[a] = 1$, $C[c] = 4$, $C[g] = 6$, and $C[t] = 7$.

i	$SA[i]$	Suffix	$T[SA[i] - 1]$
1	7	\$	g
2	1	$acacag\$$	\$
3	3	$acag\$$	c
4	5	$ag\$$	c
5	2	$cacag\$$	a
6	4	$cag\$$	a
7	6	$g\$$	a

Figure 4.21: suffix array, suffix and BW text

We now analyze the space complexity of FM-index. In the case of DNA sequence, structure 1 can be stored $2n$ bits. Structure 2 can be stored in $O(\log n)$ bit, and structure 3 can be stored in $O(\frac{n \log \log n}{\log n})$ bits. Therefore, in total the size of FM-index is $O(n)$ bits.

4.7.2 Application

One of the applications of FM-index is to determine whether a string P exists in text T , using backward search algorithm.

Input: The FM-index of T , query string P of length m .

Aim: Determine whether P exists in T .

Algorithm $BW_exist(P[1..m])$

1. $c = P[m]$, $i = m$.
2. $sp = C[c] + 1$, $ep = C[c + 1]$.
3. while ($sp \leq ep$ and $i > 1$) {
 - $c = P[i - 1]$;
 - $sp = C[c] + occ(c, 1, sp - 1) + 1$;
 - $ep = C[c] + occ(c, 1, ep)$;
 - $i = i - 1$;

4. if $sp > ep$, then pattern is not found; else the pattern is found.

Now we analyze the time complexity of backward search. To find a pattern $P[1..m]$, we need to compute $i = m$ to 1 in the while loop, which iterates $m - 1$ times. Each iteration of the loop can be computed in $O(1)$ time. Therefore, the whole backward search algorithm takes $O(m)$ time.

For example, let $T = acacag\$$, $P = aca$. Figures 4.22 to 4.24 show the computation.

First iteration,
 $P[3..3] = a$
 $sp = C[a] + 1 = 1 + 1 = 2$
 $ep = C[c] = 4$

$SA[i]$	Suffix	$T[SA[i] - 1]$
7	\$	<i>g</i>
1	<i>acacag</i> \$	\$
3	<i>acag</i> \$	<i>c</i>
5	<i>ag</i> \$	<i>c</i>
2	<i>cacag</i> \$	<i>a</i>
4	<i>cag</i> \$	<i>a</i>
6	<i>g</i> \$	<i>a</i>

Figure 4.22: first iteration of searching P in T

Second iteration,
 $P[2..3] = ca$
 $sp = C[c] + occ(c, 1, sp_{old} - 1) + 1$
 $= 4 + 0 + 1 = 5$
 $ep = C[c] + occ(c, 1, ep_{old})$
 $= 4 + 2 = 6$

$SA[i]$	Suffix	$T[SA[i] - 1]$
7	\$	<i>g</i>
1	<i>acacag</i> \$	\$
3	<i>acag</i> \$	<i>c</i>
5	<i>ag</i> \$	<i>c</i>
2	<i>cacag</i> \$	<i>a</i>
4	<i>cag</i> \$	<i>a</i>
6	<i>g</i> \$	<i>a</i>

Figure 4.23: second iteration of searching P in T

Third iteration,
 $P[1..3] = aca$
 $sp = C[a] + occ(a, 1, sp_{old} - 1) + 1$
 $= 1 + 0 + 1 = 2$
 $ep = C[a] + occ(a, 1, ep_{old})$
 $= 1 + 2 = 3$
Pattern P is found in T

$SA[i]$	Suffix	$T[SA[i] - 1]$
7	\$	<i>g</i>
1	<i>acacag</i> \$	\$
3	<i>acag</i> \$	<i>c</i>
5	<i>ag</i> \$	<i>c</i>
2	<i>cacag</i> \$	<i>a</i>
4	<i>cag</i> \$	<i>a</i>
6	<i>g</i> \$	<i>a</i>

Figure 4.24: third iteration of searching P in T

References

- [1] P. Weiner. Linear Pattern Matching Algorithms. *Switching and Automata Theory*, 1-11, 1973.
- [2] E. M. McCreight. A Space-Economical Suffix Tree Construction Algorithm. *Journal of ACM*, 23:262-272, 1976.
- [3] E. Ukkone. On-line Construction of Suffix Trees; *Algorithmica*, 14:249-260, 1995.
- [4] Martin Farah. Optimal Suffix Tree Construction with Large Alphabets. *FOCS*, 1997.
- [5] D. Harel and R. E. Tarjan. Fast Algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13:338–355, 1984.
- [6] B. Schieber and U. Vishkin. On Finding Lowest Common Ancestors: Simplifications and Parallelization. *SIAM Journal on Computing*, 17:1253–1262, 1988.
- [7] U. Manber and G. Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.
- [8] J. Larsson and K. Sadakane. Faster Suffix Sorting. Technical Report Technical Report LU-CS-TR:99-214, LUNDFD6/(NFCS-3140)/1-43/(1999), Lund University, 1999.
- [9] R. Grossi and J. S. Vitter. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. In *Proc. ACM STOC*, pages 397–406, 2000.
- [10] P. Ferragine and G. Manzini. Opportunistic Data Structures with Applications. In *Proc. IEEE FOCS*, pages 390–398, 2000.
- [11] M. Burrows and D. J. Wheeler. A Block-sorting Lossless Data Compression Algorithm. Technical Report 124, Digital Equipment Corporation, Paolo Alto, California, 1994.
- [12] Wing-Kai Hon, Kunihiro Sadakane and Wing-Kin Sung. Breaking a Time and Space Barrier in Constructing Full-Text Indices. In *Proc. IEEE FOCS*, 2003.